Teaching deductive verification with Why3 to undergraduate students

Sandrine Blazy
University of Rennes 1, CNRS IRISA, Inria

sandrine.blazy@irisa.fr

IFIP WG 2.11 Salem 2019.04.30

The students

About 100 undergraduate students, 3rd year (2nd semester)

Expected prior experience:

- introduction to functional and immutable programming (Scala, 1st year)
- introduction to team programming : modules and interfaces, test driven development, version control, contract programming (Scala, 2nd year)
- initiation to logic (propositional and predicate calculus, 3rd year)
- basic data structures (Java, 3rd year, 1st semester)

Course organisation

- Lectures: (7-2) * 2 hours
  - presentation of ideas and concepts
  - interactive demos (incl. non-trivial algorithms)
- Exercises: 8 * 2 hours
  - practice in group settings
  - prepare labs
  - quizz at the end of each session (10 minutes each)
- Labs: 10 * 2 hours
  - work in pair in small-group settings
  - submit a Why file at the end of the session (can be improved until the end of the week)
- Written exam (2 hours)

Total : 52h for each student (from January to April), mandatory course

Deductive verification in Why3

WhyML programming language: a subset of OCaml with imperative features

Several automated provers in our Linux distribution (AltErgo, cvc4, Eprover, Z3)

Many examples adapted from the Why3 gallery of verified programs

Syllabus

1. First specifications
   - **Test of specifications**
   - First programs operating over integers
     - loops, **loop invariants** and variants
     - immutability / mutable variables, let constructs
2. Type invariant
   - Arrays, sorts, matrices
3. Algebraic data types
   - **Recursive** data types (incl. lists and trees) and programs
4. **Ghost code**
5. Weakest precondition calculus

Writing formulas : hints

Many recipes are given to the students.

- to avoid bad practices
  - verbose and difficult to read formulas
    - too many variables, quantifiers,
    - big formula that should be split (e.g. a post-condition)
  - more than minimalistic formulas (e.g., the loop invariant is $0<i<N$, so that it becomes easier to prove)
- to help understand why a proof failed

Why3 is very useful !

demo: loop.mlw

Testing a specification
What is a precise specification ?

```
module Max1

val max (a b : int) : int
ensures { result=a \/ result=b }

end
```

```
module Max2

val max (a b : int) : int
ensures { result=a \/ result=b }
ensures { a ≤ result }
ensures { b ≤ result }

end
```

```
module Test

let test () =
    let m = max 3 4 in
        assert { m = 4 }
end
```

Specification: example of a sorted array

```
type elt
predicate smaller_than elt elt

predicate sorted (t : array elt) =
forall i1 i2:int. 0 ≤ i1 ≤ i2 < length t → smaller_than t[i1] t[i2]
```

9	6	12	9	2	9	9	6

2	6	6	9	9	9	9	12	12	12

```
val sort (t: array elt) : array elt
ensures { sorted result }
ensures { permut t result }
```

Testing a specification vs. testing a code

- Easily accepted by students, but sometimes difficult to assert by provers

```
let test_spec () =
    let a = make 3 0 in
        a[0] <- 7; a[1] <- 3; a[2] <- 1;
        let b = sort a in
        assert { b[0] = 1 };
        assert { b[1] = 3 };
        assert { b[2] = 7 }
```

ensures { sorted b ∧ permut a b }

Arrays, sorts, matrices : practising loop invariants

1) Basic examples where the loop invariant mimics the post-condition

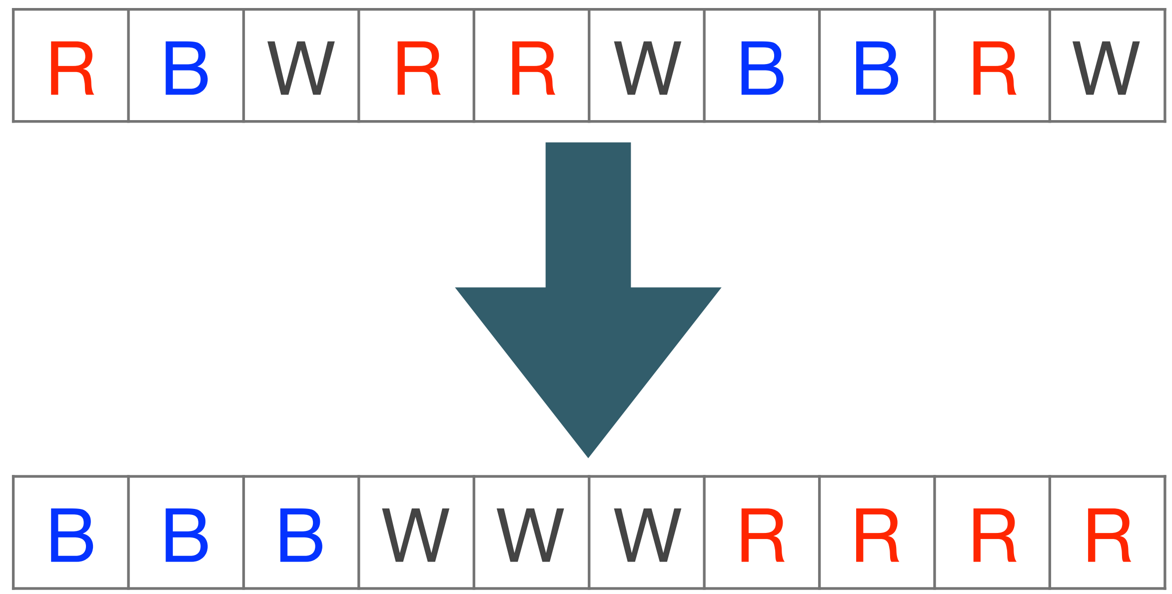
Ex.: compute the maximum of the elements of an array of natural numbers

```
predicate is_max (a: array int) (l h: int) (m: int) = …

let max_tab (a: array int) (n: int) : int
  requires { 0 ≤ n = length a }
  requires { forall i:int. 0 ≤ i < n -> a[i] >= 0 }
  ensures { is_max a 0 n result }
 =
  let m = ref 0 in
  for i = 0 to n - 1 do
    invariant {  is_max a 0 i !m }
    if !m < a[i] then m := a[i]
  done;
  !m
```

Arrays, sorts, matrices : practising loop invariants

2) Write the loop invariant first
Ex.: Dutch flag



3) Advanced examples with nested loops (e.g. insertion sort) and harder to guess invariants (e.g. selection sort, bubble sort)

Encouraging results : Why3 is very useful to find the errors of the students

Recursive programs

Programs manipulating lists and trees

- Comparison between recursive and iterative programs

- Well-known recursive programs (towers of Hanoi, a backtracking program)

- Axiomatisation of a recursive program, that is implemented using a loop

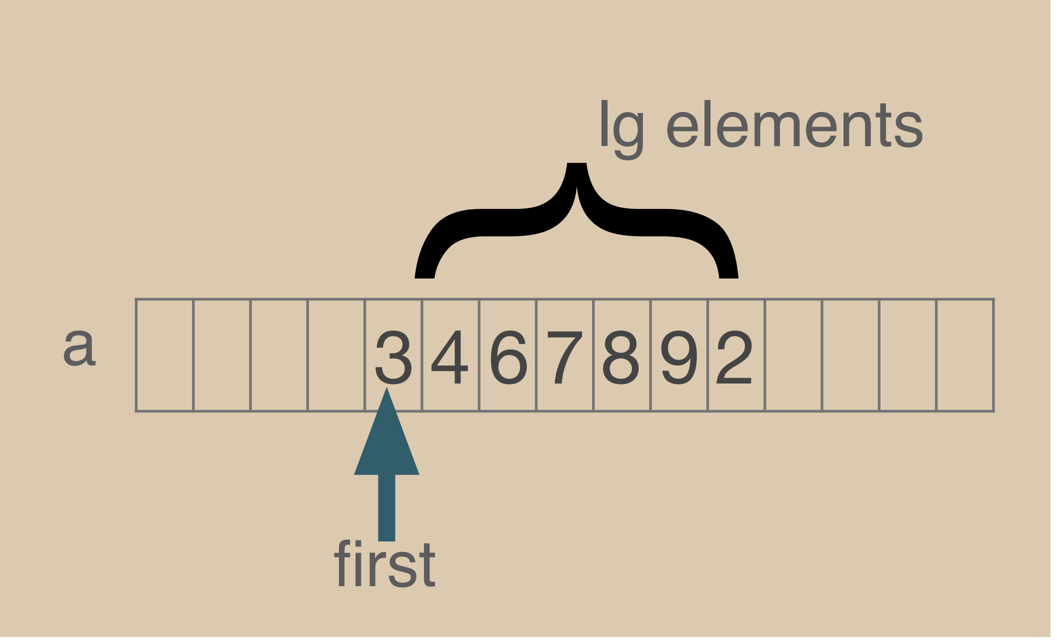
```
function fibonacci int : int
…
axiom fibn: forall n:int. n>1 ->
        fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

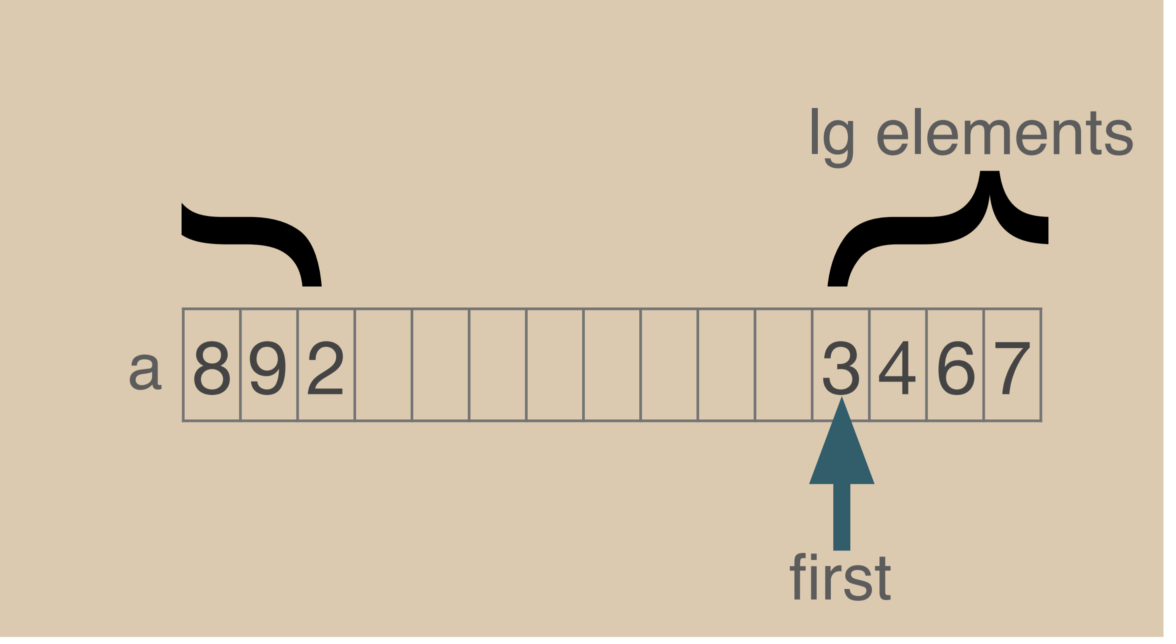
- Lab session: insertion and deletion of an element into a binary search tree

Ghost code

Use of a ghost variable and a type invariant to handle a better suited data structure

- Ex.: ring buffer





ghost sequence [3; 4; 6; 7; 8; 9; 2]

Conclusion

Understanding what is a precise specification takes some time !

- Testing a spec is very useful for beginners
- Promising feature of (some) provers : counter-example generation
- Some students tend to write a code that is not consistent with their spec
  Mandatory step : take time to think.

Recent improvements in automated provers

- Students could handle more advanced examples than expected.
- (Less and less) fragile technology: a tiny change in a spec may make it unprovable

Questions ?